A Chemistry-inspired Programming Model for Adaptive Decentralised Workflows

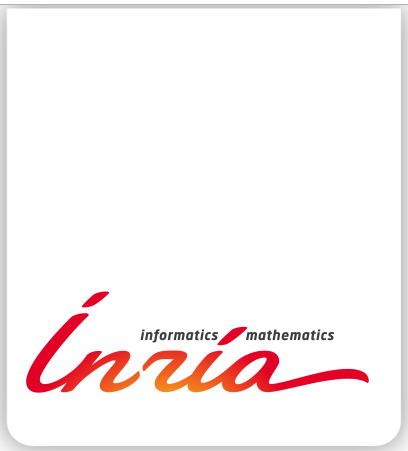
Javier Rojas Balderrama, Matthieu Simonin, Cédric Tedeschi

A Chemistry-inspired Programming Model for Adaptive Decentralised Workflows

Javier Rojas Balderrama, Matthieu Simonin, Cédric Tedeschi

RESEARCH REPORT
N° 8691
February 2015
Project-Teams Myriads

ISSN 0249-6399 ISRN INRIA/RR--8691--FR+ENG



A Chemistry-inspired Programming Model for Adaptive Decentralised Workflows

Javier Rojas Balderrama, Matthieu Simonin, Cédric Tedeschi

Project-Teams Myriads

Research Report n° 8691 — February 2015 — 16 pages

Abstract:
In this paper, we devise a chemistry-inspired programming model for the decentralised execution of scientific workflows, with the possibility of dynamically adapting its shape when its initial specification fails to reach the user's requirements or simply to run due to external conditions. We describe a decentralised architecture to support the model and cover its implementation in the *GinFlow* software prototype.

Key-words: Scientific workflows, adaptiveness, Chemical Programming Model

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Un modèle de programmation inspiré de la chimie pour des workflows décentralisés et adaptatifs

Résumé :

Dans cet article, nous présentons un modèle de programmation inspiré par les processus chimiques pour exécuter des workflows de maniére décentralisée avec la possibilité de modifier la forme du workflow lorsque le workflow initialement spécifié se montre insatisfaisant pour une raison quelconque. Nous décrivons une architecture décentralisée pour mettre en œuvre ce modéle et présentons son implémentation dans le prototype logiciel *GinFlow*.

Mots-clés : Workflows scientifiques, adapation, modèle de programmation chimique

Contents

1 Introduction

In contrast to business workflows, whose outcome is quite predictable, the output of scientific workflows (and their potential intermediate data) is rather hard to characterise prior to execution. This uncertainty comes together with the scientist's requirement to be able to explore different scenarios during computation time, due to some (possibly unsatisfactory) intermediary result. Therefore, scientific workflows are meant to support fields where variability is still largely present during experimentation.

Consequently, a workflow specification language should be able to deal with alternative workflow scenarios which will effectively get enabled only if some specific event is detected, such as a failure or the non-satisfaction of some specific property of some partial data results. This would provide a new chance to obtain meaningful results without having to start over the whole workflow enactment. At run time, this alternative workflow should be automatically triggered by the workflow engine, and in a transparent way for the user.

Another common need is scalability support. Scientists cannot afford being limited by workflow engines that may not be able to cope with workflows requiring a high overhead due to coordination at run time. Most workflow manager systems are centralised, and thus suffer from all the limitations inherent to it. However, building a decentralised engine raises several challenges related to the possibility to coordinate tasks together in a transparent way for the user.

In this paper, we tackle the general problems of finding 1) adequate programming abstractions and 2) system architectures for the development of a decentralised workflow management system with adaptiveness capabilities. Concretely, we devise the design, programming model and implementation of a decentralised workflow engine, which makes it possible to update the workflow while it is running, provided the alternative definitions to replace unsatisfactory/failed part of the workflow.

Contributions. We formulate a set of programming abstractions using a declarative chemistry-inspired model. It provides an elegant and concise way of specifying self-adaptive programs through the higher order (rules injecting new rules at run time) to define a scientific workflow, the data composition strategies, and the workflow variants declared in case of exceptions. We define two families of definitions, one for abstract workflows, to use at the user-level; another for

concrete workflows, to instantiate the abstract workflow specification and enable the adaptation rules. Finally, we review the implementation and the software technologies it relies on.
Outline. The rest of the paper is organised as follows. The programming abstractions for an adaptive decentralised workflow are detailed and exemplified in Section 2. The architecture and implementation is discussed in Section 3. The closing sections deal with related works, and conclusions.

2 Programming model

Artificial chemistries [5] provide a programming style based on the chemical metaphor. They have been shown relevant for the specification of autonomous systems. In such programming models, data are envisioned as *molecules* floating in a chemical solution, and reacting according to *reaction* rules (i.e., the program) to produce new molecules (the resulting data). Reactions are conditional, and take place between a (multi)set of molecules satisfying a reaction condition. This process continues until no reactions can be performed anymore so the solution is *inert*. Reactions take place in an implicitly parallel and autonomous way, and in a non-deterministic order. Formally, the solution is a multiset of molecules, and the possible reactions between molecules are a set of rewriting rules on this multiset. The multiset is the unique data structure, and acts similarly to an address space through which processors can cooperate. In this section, we define a set of programming abstractions for adaptive decentralised workflows. We present HOCL$_{flow}$, an extension of the HOCL language. HOCL$_{flow}$ enhances the original specification with 1) the support for some data structuring, and 2) macros to improve the clarity and conciseness of the rules.

Higher-Order Chemical Language (HOCL).

HOCL is a higher-order, rule-based declarative language inspired by a chemical metaphor and proceeding by state rewriting [1]. In HOCL, a program is a solution (i.e., a multiset) of atoms $A_1, \ldots, A_n$. An atom can be a constant (integers, booleans, etc); a tuple of atoms $e_1 : e_2 : \cdots : e_n$; a subsolution $\langle B_1, \ldots, B_m \rangle$ of atoms, or a reaction rule. A multiset of atoms is called a molecule. Note that the solution of a program is the greatest molecule of this program. A possible reaction rule between elements of the solution is written **replace-one** P **by** M **if** C, where P denotes the pattern to be satisfied by the molecules and C the condition they must satisfy. If that is the case, the reaction consumes the rule and the matched molecule, produces M, and inject it into the solution. A **replace-one** rule is one-shot: it disappears when it reacts. Its variant **replace** P **by** M **if** C is n-shot: it is not consumed when it reacts. For instance, consider a solution which calculates the maximum value out of a given set of numbers. The example below illustrates the expressiveness and higher order of HOCL, where reactions consume and/or produce other reaction rules. Let us start with the following simple program:

$$\textbf{let } \mathsf{max} = \textbf{ replace } x, y \textbf{ by } x \textbf{ if } x \geq y \textbf{ in } \langle 2, 3, 5, 8, 9, \mathsf{max} \rangle$$

The max rule consumes two integers x and y when $x \geq y$ and replaces them by x. Initially, several such reactions are possible in the provided multiset, max can use any couple of integers satisfying the condition: 2 and 3, 2 and 5, 8 and 9, etc. We may introduce a higher-order rule to only obtain the result in the final solution. Note that it is responsible to delete the max rule once the solution only contains the highest integer value. This introduces the need for sequentiality of events because we need to wait that all possible application of max to take place before deleting it. Within the chemical model, the sequentiality is achieved through subsolutions. A rule has to

wait for its inertia to access a subsolution. In our example, this leads to the encapsulation of the solution and the following program:

$$\langle\langle 2, 3, 5, 8, 9, \mathsf{max}\rangle,\ \textbf{replace-one}\ \langle \mathsf{max} = m, \omega\rangle\ \textbf{by}\ \omega\rangle$$

The m variable matches a rule named max, and ω matches all the remaining elements. One possible execution scenario within the subsolution is the following (2 and 8 on one hand, and 3 and 5 on the other hand react first, producing the intermediate state):

$$\langle 2, 3, 5, 8, 9, \mathsf{max}\rangle \rightarrow^* \langle 3, 5, 9, \mathsf{max}\rangle \rightarrow^* \langle 9, \mathsf{max}\rangle$$

Once the inertia is reached within the subsolution, the one-shot rule is triggered, extracting the result:

$$\langle\langle 9, \mathsf{max}\rangle,\ \textbf{replace-one}\ \langle \mathsf{max} = m, \omega\rangle\ \textbf{by}\ \omega\rangle \rightarrow \langle 9\rangle$$

This fine-grain example shows that HOCL provides the ability to express autonomic coordination of rules (without necessitating a centralised control). The current state of a computation is represented by the solution, which constitutes an information system by itself. The multiset becomes a shared space providing the information required for dynamic coordination, a suited requirement in a decentralised workflow engine.

2.1 HOCL$_{flow}$

We here define HOCL$_{flow}$, an extension of HOCL. The original specification supports natively very limited data structuring, through subsolutions —hierarchical structuring of unstructured subsets of elements— and tuples—ordered molecules. Tuples however can only be manipulated through pattern matching which sometimes leads to complex patterns in the left part of rules definition. We introduce lists and basic manipulation functions to improve data manipulation needed between the tasks of a workflow (see Section 2.3). Given a list ℓ and an element e, we define the following:

- () is a empty list
- list(e) creates a list containing the element e
- cons(e, ℓ) appends e to the list ℓ
- first(ℓ) returns the first element of ℓ
- rest(ℓ) returns a list whose value is ℓ without its first element
- nth(ℓ) returns the n-th element of ℓ

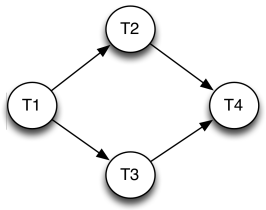
We also define the macro construct **inject**. HOCL$_{flow}$ is designed to make rules easier to read or write. In other words, it provides syntactic sugar on top of HOCL. This macro adds content into a solution without having to repeat molecules that appear in both left and right parts of the rule:

$$\textbf{inject}\ M \equiv\ \textbf{replace-one}\ \omega?\ \textbf{by}\ M, \omega$$

When a *catalyst* T is needed (i.e., an extra molecule participating in the rule activation without being consumed or modified), the **with** clause can be added to the **inject** macro along with an optional condition C:

$$\textbf{inject}\ M\ \textbf{with}\ T\ \textbf{if}\ C \equiv\ \textbf{replace-one}\ T, \omega?\ \textbf{by}\ M, T, \omega\ \textbf{if}\ C$$

where C requires the contents of T to activate the rule (e.g., the **if** clause on some element in T). Nonetheless, the with-clause can be used without any conditional when parts of contents of T are simply present in M.



$\langle$
T_1: $\langle$SRC : $\langle\rangle$, SRV : s_1, IN : $\langle$**input**$\rangle$, DST : $\langle T_2, T_3\rangle\rangle$,
T_2: $\langle$SRC : $\langle T_1\rangle$, SRV : s_2, IN : $\langle\rangle$, DST : $\langle T_4\rangle\rangle$,
T_3: $\langle$SRC : $\langle T_1\rangle$, SRV : s_3, IN : $\langle\rangle$, DST : $\langle T_4\rangle\rangle$,
T_4: $\langle$SRC : $\langle T_2, T_3\rangle$, SRV : s_4, IN : $\langle\rangle$, DST : $\langle\rangle\rangle$
$\rangle$

Figure 1: Workflow DAG (left), HOCL$_{flow}$ definition (right)

2.2 Tasks, bindings and dependencies

There are two fundamental information without which a workflow would not exist and which are, consequently, required by any workflow runtime system. Firstly, the binding to actual services performing the tasks; and secondly, the links connecting these tasks together (e.g., the successors and predecessors for the task). Let us take a simple workflow example and review its HOCL$_{flow}$ representation, as illustrated by Figure 1.

A workflow is composed of as many subsolutions as tasks in a DAG definition. Each subsolution contains binding information and dependencies for one task. The atom SRV declares the binding to the actual service s_i. The molecule IN : $\langle\rangle$ contains the parameters required to invoke it, and molecules SRC : $\langle\rangle$ and DST : $\langle\rangle$ the source and destination dependencies, respectively. For instance, T_2 requires the results of T_1 as input, and its results will be sent to T_4.

A workflow definition needs to get set into action, since the above HOCL code is a mere description of *what to do* (not how to do it). Any workflow engine needs to 1) call services, and 2) transfer data between them. Rules are *re-writers*, as previously explained, so they can change the workflow state expressed by the HOCL$_{flow}$ multiset. The application of such rules allow us to reflect the completion of a task or the availability of partial results. Hence, rules should be *generic* to process any HOCL$_{flow}$ workflow definition. In order to achieve those two requirements, three generic workflow rules (gw_∗) are defined below:

$$\begin{aligned}\textbf{let } \mathsf{gw\_setup} = \textbf{replace-one } &\text{SRC} : \langle\rangle, \text{IN} : \langle\omega_{\text{IN}}?\rangle \\ \textbf{by } &\text{SRC} : \langle\rangle, \text{PAR} : \ell_{\text{PAR}} = \text{list}(\omega_{\text{IN}}), \text{RES} : ()\end{aligned} \tag{1}$$

$$\begin{aligned}\textbf{let } \mathsf{gw\_call} = \textbf{replace-one } &\text{SRC} : \langle\rangle, \text{SRV} : s_i, \text{PAR} : \ell_{\text{PAR}}, \text{RES} : \ell_{\text{RES}} \\ \textbf{by } &\text{SRC} : \langle\rangle, \text{SRV} : s_i, \text{PAR} : \text{rest}(\ell_{\text{PAR}}), \\ &\text{RES} : \text{cons}(\texttt{invoke}(s_i, \text{first}(\ell_{\text{PAR}})), \ell_{\text{RES}}) \textbf{ if } \ell_{\text{PAR}} \neq ()\end{aligned} \tag{2}$$

$$\begin{aligned}\textbf{let } \mathsf{gw\_pass} = \textbf{replace } &T_i : \langle\text{PAR} : \ell_{\text{PAR}}, \text{RES} : \ell_{\text{RES}}, \text{DST} : \langle T_j, \omega_{\text{DST}}?\rangle, \omega_i?\rangle, \\ &T_j : \langle\text{SRC} : \langle T_i, \omega_{\text{SRC}}?\rangle, \text{IN} : \langle\omega_{\text{IN}}?\rangle, \omega_j?\rangle \\ \textbf{by } &T_i : \langle\text{PAR} : \ell_{\text{PAR}}, \text{RES} : \ell_{\text{RES}}, \text{DST} : \langle\omega_{\text{DST}}\rangle, \omega_i\rangle, \\ &T_j : \langle\text{SRC} : \langle\omega_{\text{SRC}}\rangle, \text{IN} : \langle\ell_{\text{RES}}, \omega_{\text{IN}}\rangle, \omega_j\rangle \textbf{ if } \ell_{\text{PAR}} = ()\end{aligned} \tag{3}$$

Rule gw_setup detects the readiness of the task (i.e., all dependencies have been satisfied) through the emptiness of the SRC : $\langle\rangle$ molecule. Upon application, the rule moves the content from the molecule IN : $\langle\rangle$ to the atom PAR as a list of parameters. It also creates the atom RES as an empty list which, later, will contain the results of the service invocation. These molecules altogether trigger the gw_call which invokes the service implementing the task. After service

```
4.01 ⟨ gw_pass,
4.02   T1 : ⟨SRC : ⟨⟩, SRV : s1, IN : ⟨input⟩, DST : ⟨T2, T3⟩, gw_setup, gw_call⟩,
4.03   T2 : ⟨SRC : ⟨T1⟩, SRV : s2, IN : ⟨⟩, DST : ⟨T4⟩, gw_setup, gw_call⟩,
4.04   T3 : ⟨SRC : ⟨T1⟩, SRV : s3, IN : ⟨⟩, DST : ⟨T4⟩, gw_setup, gw_call⟩,
4.05   T4 : ⟨SRC : ⟨T2, T3⟩, SRV : s4, IN : ⟨⟩, DST : ⟨⟩, gw_setup, gw_call⟩⟩
```

Figure 2: Concrete workflow (abstract workflow + generic workflow rules)

completion, the result is inserted in RES. This rule is applied as long as elements are available in PAR. At first sight, IN : $\langle\rangle$ and PAR seem to be redundant, however, IN : $\langle\rangle$ is a container for results received from sources, whereas PAR is a list buffer of parameters for the invocation of services. Section 2.3 describes their different purpose and usage in greater detail. In practice, rules gw_setup and gw_call are used together. They are internal in the sense that they have to be present in each subsolution (see Figure 2).

Rule gw_pass is responsible for transferring results from a source to forthcoming destinations. It is global to the workflow because its scope is not limited to one subsolution. Its scope spans at least two services (one source and one destination). gw_pass is triggered after the result has been obtained and placed in RES. It moves the resulting value from the source as inputs to each declared destination. Then it updates the sources and destinations to reflect that the dependency is satisfied. This rule may be triggered as many times as necessary according to all dependencies defined in molecules SRC : $\langle\rangle$ and DST : $\langle\rangle$.

These generic rules constitute a high-level programmatic way to enact a workflow expressed with HOCL$_{flow}$. In means that an abstract workflow is changed into a concrete one by adding these rules. Once the concrete workflow is obtained, it can be run by the HOCL interpreter. The concrete workflow corresponding to the workflow in Figure 1 is provided in Figure 2.

2.3 Data composition strategies

It is common that a task does not receive a simple result from its sources but a data *set*. How to combine the elements of these sets into distinct inputs to the service realising the tasks relies on *composition strategies*. In this section, we introduce the HOCL$_{flow}$ mechanisms to process some of the most common strategies described in literature [10], namely the dot and cross products, and the input filtering. In the following, let us assume tasks have results provided as lists, for example, $res_i = (a_0, a_1, \ldots, a_{m-1})$, and $res_j = (b_0, b_1, \ldots, b_{n-1})$.

Dot product.

The dot product is the set of ordered pairs (a_k, b_k) for all k such that $0 \leq k < \min(m, n)$ from the resulting combination of the elements of res_i and res_j represented as lists ℓ_i and ℓ_j respectively. Each pair serves as an independent input for the invocation of the service supporting the task. In HOCL$_{flow}$, the dot product (dp_$*$) can be defined through rules 4 and 5.

$$\begin{aligned}\textbf{let } \mathsf{dp\_cons} = \textbf{replace } &\text{SRC} : \langle \text{DOT} \rangle, \text{IN} : \langle \ell_i, \ell_j \rangle, \text{PAR} : \ell_{\text{PAR}} \\ \textbf{by } &\text{SRC} : \langle \text{DOT} \rangle, \text{IN} : \langle \text{rest}(\ell_i), \text{rest}(\ell_j) \rangle, \\ &\text{PAR} : \text{cons}((\text{first}(\ell_i) : \text{first}(\ell_j)), \ell_{\text{PAR}}) \textbf{ if } \ell_i \neq () \wedge \ell_j \neq ()\end{aligned} \tag{4}$$

$$\textbf{let } \mathsf{dp\_setup} = \textbf{replace-one } \textsc{src} : \langle \textsc{dot} \rangle, \textsc{in} : \langle \ell_i, \ell_j \rangle \textbf{ by } \textsc{src} : \langle \rangle, \textsc{res} : () \textbf{ if } \ell_i = () \vee \ell_j = () \tag{5}$$

At run time, the rule dp_cons is applied as soon as the required molecules are available. It creates the pairs for the service invocation in the PAR atom. This allows the interpreter to apply the rule gw_call, as many times as necessary—once for each pair. Consequently, the rule dp_setup can be applied when processed molecules are empty, paving the way for the application of the gw_call rule with the created pairs, again once for each pair.

Cross Product.

The cross product is the set of all ordered pairs (a_k, b_l) with $0 \leq k < m$ and $0 \leq l < n$. Each pair serves as an independent input for an invocation of the service supporting the task. In HOCL$_{flow}$, the cross product (cp_$*$) can be defined through rules 6, 7, 8, and 9.

$$\textbf{let } \mathsf{cp\_init} = \textbf{inject } \textsc{in}' : \langle \ell_{\textsc{backup}} = \ell_j \rangle \textbf{ with } \textsc{in} : \langle \ell_i, \ell_j \rangle \tag{6}$$

$$\textbf{let } \mathsf{cp\_cons} = \textbf{replace } \textsc{src} : \langle \textsc{cross} \rangle, \textsc{in} : \langle \ell_i, \ell_j \rangle, \textsc{par} : \ell_{\textsc{par}} \textbf{ by } \textsc{src} : \langle \textsc{cross} \rangle, \textsc{in} : \langle \ell_i, \mathrm{rest}(\ell_j) \rangle, \textsc{par} : \mathrm{cons}((\mathrm{first}(\ell_i) : \mathrm{first}(\ell_j)), \ell_{\textsc{par}}) \textbf{ if } \ell_j \neq () \tag{7}$$

$$\textbf{let } \mathsf{cp\_next} = \textbf{replace } \textsc{src} : \langle \textsc{cross} \rangle, \textsc{in} : \langle \ell_i, \ell_j \rangle, \textsc{in}' : \langle \ell_{\textsc{backup}} \rangle \textbf{ by } \textsc{src} : \langle \textsc{cross} \rangle, \textsc{in} : \langle \mathrm{rest}(\ell_i), \ell_{\textsc{backup}} \rangle, \textsc{in}' : \langle \ell_{\textsc{backup}} \rangle \textbf{ if } \ell_i \neq () \wedge \ell_j = () \tag{8}$$

$$\textbf{let } \mathsf{cp\_setup} = \textbf{replace-one } \textsc{src} : \langle \textsc{cross} \rangle, \textsc{in} : \langle \ell_i, \ell_j \rangle, \textsc{in}' : \langle \ell_{\textsc{backup}} \rangle \textbf{ by } \textsc{src} : \langle \rangle, \textsc{res} : () \textbf{ if } \ell_i = () \tag{9}$$

The cross product is handled in a way similar to the dot product. Firstly, the cp_init rule creates a backup copy of the ℓ_j list, by injecting a $\textsc{in}' : \langle \rangle$ molecule. This copy is used after each element in the list ℓ_i has been combined with all of the elements in ℓ_j, to *refill* $\textsc{in} : \langle \rangle$ so it can be combined with the next element in ℓ_i. Then, cp_cons iterates through ℓ_j while the element of ℓ_i is fixed, removing the element from ℓ_j at each iteration. Once ℓ_j is empty, the rule cp_next can be applied to refill ℓ_j and remove the element of ℓ_i that was just processed. The rule cp_setup has the same role as the homologous rule dp_setup.

Filtering.

A composite result may be composed of several parts having different purposes. In workflows, this is commonly dealt with through data filtering: when the result reaches one of its destination, it goes through a filter so only the relevant part in the source is selected. The filtering (df_$*$) can be defined through rules 10 and 11.

$$\textbf{let } \mathsf{df\_init} = \textbf{replace-one } \textsc{src} : \langle \textsc{filter} : (T_i : j), \omega_{\textsc{src}}? \rangle, \textsc{in} : \langle \ell_i, \omega_{\textsc{in}}? \rangle \textbf{ by } \textsc{src} : \langle \textsc{filter} : (T_i : j), \omega_{\textsc{src}} \rangle, \textsc{in} : \langle \omega_{\textsc{in}} \rangle, \textsc{in}' : \langle \ell_i \rangle \tag{10}$$

$$\textbf{let } \mathsf{df\_step} = \textbf{replace } \textsc{src} : \langle \textsc{filter} : (T_i : j), \omega_{\textsc{src}}? \rangle, \textsc{in} : \langle \omega_{\textsc{in}}? \rangle, \textsc{in}' : \langle \ell_i \rangle \textbf{ by } \textsc{src} : \langle \omega_{\textsc{src}} \rangle, \textsc{in} : \langle \mathrm{nth}(j, \ell_i), \omega_{\textsc{in}} \rangle, \textsc{in}' : \langle \ell_i \rangle \tag{11}$$

6.01 $\langle$ gw_pass,
6.02 $T_1 : \langle \text{SRC} : \langle\rangle, \text{SRV} : s_1, \text{IN} : \langle \texttt{input} \rangle, \text{DST} : \langle T_2, T_3 \rangle, \mathsf{gw\_setup}, \mathsf{gw\_call} \rangle$,
6.03 $T_2 : \langle \text{SRC} : \langle \text{FILTER} : (T_1 : 3), T_1 \rangle, \text{SRV} : s_2, \text{IN} : \langle\rangle, \text{DST} : \langle T_4 \rangle,$
6.04 $\mathsf{df\_init}, \mathsf{df\_step}, \mathsf{gw\_setup}, \mathsf{gw\_call} \rangle$,
6.05 $T_3 : \langle \text{SRC} : \langle T_1 \rangle, \text{SRV} : s_3, \text{IN} : \langle\rangle, \text{DST} : \langle T_4 \rangle, \mathsf{gw\_setup}, \mathsf{gw\_call} \rangle$,
6.06 $T_4 : \langle \text{SRC} : \langle \text{DOT}, T_2, T_3 \rangle, \text{SRV} : s_4, \text{IN} : \langle\rangle, \text{DST} : \langle\rangle, \mathsf{dp\_setup}, \mathsf{dp\_cons}, \mathsf{gw\_call} \rangle \rangle$

Figure 3: Concrete workflow with data pre-processing

At the abstract workflow level, expressing a filter requires adding a molecule $\text{FILTER} : (T_i, j)$ on $\text{SRC} : \langle\rangle$. The subsolution T_i denotes the origin of the composite result, and j gives the rank of the element needed in ℓ_i. Each result can be filtered at several positions, with as many $\text{FILTER} : (T_i, *)$ molecules as necessary to filter ℓ_i. The df_init rule isolates the ℓ_i list provided a molecule $\text{FILTER} : (T_i, *)$ is found. Note that even if several molecules $\text{FILTER} : (T_i, *)$ are present, performing this isolation once is enough. Then, the rule df_step can be triggered as many times as necessary to actually keep only the requested parts.

Example.

Let us consider the simple workflow described in Figure 3 to illustrate the building blocks of our programming model presented above. This workflow has the same structure as the workflow depicted previously in Figure 1. Its difference stands in the sense that it locally pre-processes data prior to some of services' invocation. Specifically, it applies a filter on the result of T_1 before submitting it to Service s_2. A dot product is also applied on the results of T_2 and T_3 before invoking T_4. Note that only the relevant generic rules were inserted. The details of the generation of the concrete workflow are described in Section 3. We detail next the execution of the workflow focusing on the rewriting of the multiset. For the sake of clarity, the generic rules are omitted in the multiset, so only the abstract workflow is represented. Initially, only the gw_setup rule is enabled, in subsolution T_1. It is triggered to prepare the service invocation. An atom PAR is created, rewriting the subsolution T_1:

$$T_1 : \langle \text{SRC} : \langle\rangle, \text{SRV} : s_1, \text{PAR} : (\texttt{input}), \text{DST} : \langle T_2, T_3 \rangle \rangle$$

Again, only the gw_call rule can be applied. Triggering it will actually call the service and inject the result in Subsolution T_1:

$$T_1 : \langle \text{SRC} : \langle\rangle, \text{SRV} : s_1, \text{PAR} : (), \text{RES} : (res_1), \text{DST} : \langle T_2, T_3 \rangle \rangle$$

Heretofore, subsolutions other than T_1 have been inert: no combinations of molecules in them could match one rule's left part. The outside rule gw_pass is now enabled; the results of T_1 are available and they can be sent to its destinations, namely T_2 and T_3. Note that the gw_pass rule is triggered twice, once for each destination of T_1, adding to $\text{IN} : \langle\rangle$ the respective res_1. At this point, subsolution T_1 is inert definitively. All these subsolutions are then updated, reflecting the transfer and leading to the following multiset:

$$T_1 : \langle \text{SRC} : \langle\rangle, \text{SRV} : s_1, \text{PAR} : (), \text{RES} : (res_1), \text{DST} : \langle\rangle \rangle$$
$$T_2 : \langle \text{SRC} : \langle \text{FILTER} : (T_1 : 3) \rangle, \text{SRV} : s_2, \text{IN} : \langle res_1 \rangle, \text{DST} : \langle T_4 \rangle \rangle$$
$$T_3 : \langle \text{SRC} : \langle\rangle, \text{SRV} : s_3, \text{IN} : \langle res_1 \rangle, \text{DST} : \langle T_4 \rangle \rangle$$
$$T_4 : \langle \text{SRC} : \langle \text{DOT}, T_2, T_3 \rangle, \text{SRV} : s_4, \text{IN} : \langle\rangle, \text{DST} : \langle\rangle \rangle$$

It is now possible to process T_2 and T_3 in parallel. In T_3, rules gw_setup and gw_call can be triggered, one after another. However, due to the presence of FILTER : $(T_1 : 3)$ inside SRC : $\langle\rangle$, Rule gw_setup is not enabled in T_2 because it requires an empty molecule SRC : $\langle\rangle$. That same atom will first trigger the df_init and df_step rules, extracting the filtered part of the result. At the same time, SRC : $\langle\rangle$ is flushed, and a PAR atom is created. Therefore, the subsolution T_2 is now ready for the application of the rules gw_setup and gw_call:

$$T_2 : \langle \text{SRC} : \langle\rangle, \text{SRV} : s_2, \text{IN} : \langle res_{13} \rangle, \text{DST} : \langle T_4 \rangle \rangle$$

Once gw_setup and gw_call have been applied within subsolutions T_2 and T_3, gw_pass can transfer results into IN : $\langle\rangle$ in T_4, as previously from T_1 to T_2 and T_3. Subsolutions T_2 and T_3 are now definitively inert:

$$T_1 : \langle \text{SRC} : \langle\rangle, \text{SRV} : s_1, \text{PAR} : (), \text{RES} : (res_1), \text{DST} : \langle\rangle \rangle$$
$$T_2 : \langle \text{SRC} : \langle\rangle, \text{SRV} : s_2, \text{PAR} : (), \text{RES} : (res_2), \text{DST} : \langle\rangle \rangle$$
$$T_3 : \langle \text{SRC} : \langle\rangle, \text{SRV} : s_3, \text{PAR} : (), \text{RES} : (res_3), \text{DST} : \langle\rangle \rangle$$
$$T_4 : \langle \text{SRC} : \langle \text{DOT} \rangle, \text{SRV} : s_4, \text{IN} : \langle (res_{21} : res_{22}), (res_{31} : res_{32}), \text{DST} : \langle\rangle \rangle$$

Now dp_cons can be applied and the pairs are created by its repeated application. Let us assume, for instance, that the results of subsolutions T_2 and T_3 are made of two elements each, denoted res_{21}, res_{22} and res_{31}, res_{32} respectively. These results in the subsolution T_4 are defined as following:

$$T_4 : \langle \text{SRC} : \langle \text{DOT} \rangle, \text{SRV} : s_4, \text{IN} : \langle\rangle, \text{PAR} : ((res_{21} : res_{31}), (res_{22} : res_{32})), \text{DST} : \langle\rangle \rangle$$

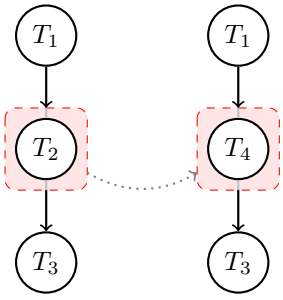
Once dp_setup has been applied, cleaning SRC : $\langle\rangle$, gw_call is triggered for each pair of PAR, invoking s_4 twice, and obtaining the two final results. This leads to the final state of all workflow and an inert multiset.

2.4 Adaptiveness

Exception-handling support during workflow execution is typically managed at infrastructure level. Autonomic systems adapt to exceptional situations while hiding their intrinsic complexity to users. In scientific workflows, those exceptions (e.g., unavailability, software error or middleware failures) can be detected by monitoring the execution status of the services and providing corrective actions using pre-defined policies. An alternative approach in programmatic environments may be to declare the adaptiveness in an explicit manner. Such a programmatic adaptiveness provides mechanisms to adapt the workflow itself when facing changing conditions while letting users the possibility to express the way the workflow enactment evolves. Similarly to the basic workflow treatment, the user-defined abstract adaptive workflow needs to be extended becoming a concrete workflow as executable HOCL code.

We identify four steps to implement workflow adaptiveness as an extension of the set of rules defining the workflow engine. These are: 1) the declaration of alternatives to replace failing services 2) the identification of the exception, 3) the propagation of the exception to the concerned dependent services, and 4) replacement of concerned services with their equivalent, replacement services. The user is responsible for the inclusion of an alternative scenario in the workflow definition. An alternative workflow excerpt must be either a simple service or a composite service, but with only one exit route. This condition is set to avoid incoherent states during the enactment. A workflow example illustrating the adaptiveness is given in Figure 4.

The adaptation strategy is declared in an abstract workflow by defining the subsolution(s) associated with the alternative service(s). The main idea is to rely on an inline rule that, combined



7.01 $\langle\ T_1 : \langle \text{SRC} : \langle\rangle, \text{SRV} : s_1, \text{IN} : \langle \texttt{input} \rangle, \text{DST} : \langle T_2 \rangle\rangle,$
7.02 $\quad T_2 : \langle \text{SRC} : \langle T_1 \rangle, \text{SRV} : s_2, \text{IN} : \langle\rangle, \text{DST} : \langle T_3 \rangle\rangle,$
7.03 $\quad T_3 : \langle \text{SRC} : \langle T_2 \rangle, \text{SRV} : s_3, \text{IN} : \langle\rangle, \text{DST} : \langle\rangle\rangle,$

7.04 $\quad T_4 : \langle \text{SRC} : \langle T_1 \rangle, \text{SRV} : s_4, \text{IN} : \langle\rangle, \text{DST} : \langle T_3 \rangle\rangle,$
7.05 $\quad \textbf{inject}\ \text{TRIGGER} : T_4, \text{BOND} : \langle (T_1 : T_3) \rangle$
7.06 $\qquad\qquad \textbf{with}\ T_2 : \langle \text{RES} : (\omega_{\text{RES}}), \omega_2 \rangle\ \textbf{if}\ \omega_{\text{RES}} = \texttt{failed}\ \rangle$

Figure 4: Adaptiveness: Schema (left) and abstract workflow (right)

with the replacing service(s), starts the adaptive process. This rule includes the atoms TRIGGER and BOND defining the replacement, the new bindings between services, and the exception condition. It is applied as soon as the propagation of an exception is detected. Atoms TRIGGER and BOND can be seen as special meta-clauses that will be replaced, during the generation of the concrete workflow out of the abstract one, with parametric rules to be enacted if the alternate workflow needs to get started. The TRIGGER atom refers to the alternative services that may be used in case of another service raises an exception. The BOND atom should specify where the adaptiveness needs to get propagated, so as to link the new dependencies (i.e., recreating the chemistry bonds between subsolutions). The inline rule has to declare as many atoms of type BOND as dependencies are declared in the sources and destinations of the related subsolutions. This is all the information users need to define.

Let us review the workflow adaptiveness generic or generated rules (wa_$*$) that will be added to the user-defined workflow so as to enable the exception handling with high-level definition prior to execution. They are defined below:

$$\textbf{let}\ \mathsf{wa\_init} = \textbf{inject}\ \text{DST}' : \langle\rangle \tag{12}$$

$$\begin{aligned}\textbf{let}\ \mathsf{wa\_pass} = \textbf{replace}\ & T_i : \langle \text{PAR} : \ell_{\text{PAR}}, \text{RES} : \ell_{\text{RES}}, \text{DST} : \langle T_j, \omega_{\text{DST}}? \rangle, \text{DST}' : \langle \omega_{\text{DST}'}? \rangle, \omega_i? \rangle, \\ & T_j : \langle \text{SRC} : \langle T_i, \omega_{\text{SRC}}? \rangle, \text{IN} : \langle \omega_{\text{IN}}? \rangle, \omega_j? \rangle \\ \textbf{by}\ & T_i : \langle \text{PAR} : \ell_{\text{PAR}}, \text{RES} : \ell_{\text{RES}}, \text{DST} : \langle \omega_{\text{DST}} \rangle, \text{DST}' : \langle T_j, \omega_{\text{DST}'} \rangle, \omega_i \rangle, \\ & T_j : \langle \text{SRC} : \langle \omega_{\text{SRC}} \rangle, \text{IN} : \langle \ell_{\text{RES}}, \omega_{\text{IN}} \rangle, \omega_j \rangle \\ \textbf{if}\ & \ell_{\text{PAR}} = () \wedge \texttt{failed} \notin \ell_{\text{RES}}\end{aligned} \tag{13}$$

$$\textbf{let}\ \mathsf{wa\_update\text{-}SRC}_{ij} = \textbf{replace-one}\ \text{SRC} : \langle T_i, \omega_{\text{SRC}}? \rangle\ \textbf{by}\ \text{SRC} : \langle T_j, \omega_{\text{SRC}} \rangle \tag{14}$$

$$\begin{aligned}\textbf{let}\ \mathsf{wa\_update\text{-}DST}_{ij} = \textbf{replace-one}\ & \text{DST} : \langle \omega_{\text{DST}}? \rangle, \text{DST}' : \langle T_i, \omega_{\text{DST}'}? \rangle \\ \textbf{by}\ & \text{DST} : \langle T_j, \omega_{\text{DST}} \rangle, \text{DST}' : \langle \omega_{\text{DST}'} \rangle\end{aligned} \tag{15}$$

$$\begin{aligned}\textbf{let}\ \mathsf{wa\_trigger\text{-}DST}_{kij} = \textbf{replace-one}\ & T_k : \langle \omega_k? \rangle \\ \textbf{by}\ & T_k : \langle \mathsf{wa\_update\text{-}SRC}_{ij}, \omega_k \rangle\end{aligned} \tag{16}$$

$$\begin{aligned}\textbf{let}\ \mathsf{wa\_trigger\text{-}SRC}_{kij} = \textbf{replace-one}\ & T_k : \langle \omega_k? \rangle \\ \textbf{by}\ & T_k : \langle \mathsf{wa\_update\text{-}DST}_{ij}, \omega_k \rangle\end{aligned} \tag{17}$$

The wa_init rule injects, into each subsolution, the $\text{DST}' : \langle\rangle$ molecule. This molecule acts as a supplementary holder of sources identified as having an alternative service in case of exception. Then the wa_pass rule, an extended version of its gw_pass analogous, not only transfers results from sources to destinations when the service invocation is achieved properly, but also updates all dependencies associated to the service. It replaces the service with the alternative declared in the excerpt associated to the workflow adaptiveness.

Rules wa_update-$*$ and wa_trigger-$*$ represent the core instrumentation of the workflow adaptiveness. Unlike all previous rules, this set of rules is *parametric*; they incorporate, in their definition, variables indexed i, j, k to refer to the involved subsolutions. They are generated specifically for each workflow based on the definition of the abstract workflow. Rules wa_update-$*$ update the dependency bindings of the subsolutions changing the contents of sources and destinations. The wa_update-SRC_{ij} rule updates the source T_i with the alternative T_j in all subsolutions containing this dependency. In the same way, wa_update-DST_{ij} replaces the old references of the destination T_i with the value T_j.

Finally, rules wa_trigger-$*$, which take advantage of the high-order properties, adapt the workflow definition, enabling the new subsolution as a proper part of the workflow. The wa_trigger-DST_{kij} rule amends the contents of subsolution T_i containing T_k as part of their destinations with the new reference T_j. This process resets the inert status of already invoked services, by forcing to transfer its results to their newly added destinations, as defined in Rule wa_pass. In a similar way, the wa_trigger-SRC_{kij} rule removes the references to the services which raised an exception, replacing them by the alternative subsolution. Following the example of Figure 4, the resulting concrete workflow will include the following excerpt:

$$
\begin{aligned}
&T_4 : \langle \text{SRC} : \langle T_1 \rangle, \text{SRV} : s_4, \text{IN} : \langle\rangle, \text{DST} : \langle T_3 \rangle, \mathsf{wa\_init}, \mathsf{gw\_setup}, \mathsf{gw\_call} \ \rangle \\
&\textbf{inject } \mathsf{wa\_trigger\text{-}DST}_{324}, \mathsf{wa\_trigger\text{-}SRC}_{124} \\
&\qquad \textbf{with } T_2 : \langle \text{RES} : (\omega_{\text{RES}}), \omega_2 \rangle \textbf{ if } \omega_{\text{RES}} = \texttt{failed}
\end{aligned}
$$

At time of *concretisation*, the subsolution and the inline adaptive rule will be used to generate the following set of rules:

$$
\begin{aligned}
&\textbf{let } \mathsf{wa\_trigger\text{-}DST}_{324} = \textbf{ replace-one } T_3 : \langle \omega_3 ? \rangle \textbf{ by } T_3 : \langle \mathsf{wa\_update\text{-}SRC}_{24}, \omega_3 \rangle \\
&\textbf{let } \mathsf{wa\_trigger\text{-}SRC}_{124} = \textbf{ replace-one } T_1 : \langle \omega_1 ? \rangle \textbf{ by } T_1 : \langle \mathsf{wa\_update\text{-}DST}_{24}, \omega_1 \rangle \\
&\textbf{let } \mathsf{wa\_update\text{-}SRC}_{24} = \textbf{ replace-one } \text{SRC} : \langle T_2, \omega_{\text{SRC}} ? \rangle \textbf{ by } \text{SRC} : \langle T_4, \omega_{\text{SRC}} \rangle \\
&\textbf{let } \mathsf{wa\_update\text{-}DST}_{24} = \textbf{ replace-one } \text{DST} : \langle \omega_{\text{DST}} ? \rangle, \text{DST}' : \langle T_2, \omega_{\text{DST}'} ? \rangle \\
&\qquad\qquad\qquad\qquad \textbf{by } \text{DST} : \langle T_4, \omega_{\text{DST}} \rangle, \text{DST}' : \langle \omega_{\text{DST}'} \rangle
\end{aligned}
$$

In general, a failing service may be replaced with another equivalent service (or set of services). This replacement does not require any extra runtime procedures at the system level. The parametric rules define all the actions to be taken (provided an HOCL_{flow} interpreter). The final definition of the adaptive workflow does not modify the original workflow definition. All the complexity of the enactment instrumentation is also hidden from the user.

3 Architecture and implementation

In this section, we describe the design principles of GinFlow, the software prototype developed to implement the proposed programming model in a decentralised fashion. As we explained previously, decentralisation is one of our main motivations. The architecture proposed takes its roots in [6], and relies on a shared-space based coordination model, as depicted in Figure 5. We assume each service is deployed and encapsulated into a Service Agent (SA). SAs communicate

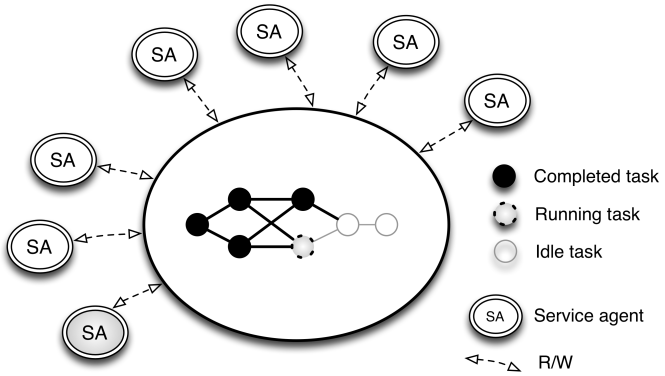


Figure 5: A decentralised workflow execution environment

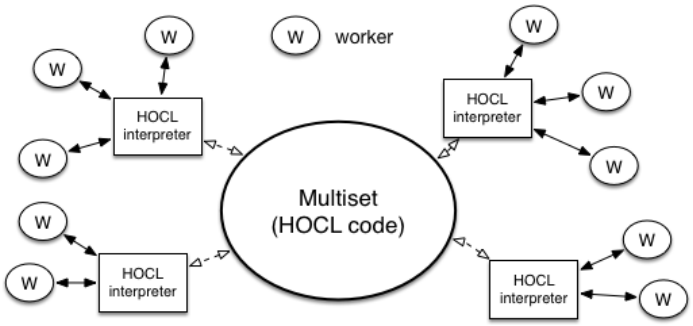


Figure 6: Ginflow decentralised architecture

through the shared space, which contains a description of (the state of) the workflow. This description is assumed to reflect the progress of the workflow, in particular completed tasks, and available input data. Each SA can rely on the space to act. The service agents execute one task, and take part in the coordination by reading and writing the shared space. Doing so, they notify when agents of subsequent executions can start (provided other incoming dependencies have also been satisfied).

3.1 The GinFlow prototype

Figure 6 shows a logical view of the GinFlow architecture. As mentioned before, the initial workflow is described as an HOCL program. The shared space is implemented through the multiset containing it. To implement the SAs, GinFlow depends on a set of HOCL interpretors acting as co-engines and a set of workers responsible for the actual execution of services. GinFlow may deploy as many co-engines as tasks in the workflow. Each co-engine concurrently read the multiset to collect (and copy locally) the status of the workflow. Each embedded HOCL engine can therefore detect if some process is needed.

For instance, when a dot product is computed on a co-engine, it iterates over the inputs as shown in Section 2.3 and produces a list of parameters. Then, it triggers the execution of

the service, wrapped in one or several workers in parallel, depending on the size of the input parameters list. In other words, each time the gw_call rule is triggered, a new worker receives work. Finally, the co-engine collects the results and pushes them back, along with the adapted HOCL description of the new workflow status, into the shared multiset. The co-engines lay at the interface between the HOCL domain and the services. They act as local coordinators for the execution of the HOCL service described in the multiset through the workers. These workers are generic in the sense of execution because they can call different implementations (SOAP, REST, binaries, etc). They are handled through a common abstraction layer. Moreover they are lightweight, stateless and fully decoupled from the other components of the architecture. In the current implementation, the HOCL core language, and the messaging protocol inherent to the distributed nature of the implementation are independent modules. GinFlow relies on the ActiveMQ middleware to implement the communications between the multiset and the HOCL engines (i.e., the gw_pass rule is implemented through ActiveMQ queues).

3.2 Concrete workflow generation and enactment

Section 2.1 describes how abstract workflows are translated into concrete workflows prior to execution. The user only has to specify the source and destination for each service. The pre-processing phase will generate what is needed for the workflow to run correctly (passing molecules from one service to another, iterating over the inputs, and so on). Thus the compilation of an HOCL_{flow} workflow follows a pipeline of operations that consists in 1) parsing the abstract description and adding the generic rules allowing service invocation, data pre-processing and transfer, 2) generating the concrete rules from meta-molecules (e.g BOND and TRIGGER), and 3) issuing the resulting workflow to the HOCL_{flow} compiler.

The pre-processing steps (1) and (2) parse the input abstract workflow and add the relevant rules to the workflow. Following the same principle, specific rules supporting the adaptiveness are hidden from the user (see Figure 4). In order to implement this process, we rely on a mechanism similar to the one which enables the molecule transfer between services (using the gw_pass rule). Such communications between HOCL entities make use of the publish/subscribe capabilities of the ActiveMQ broker. Figure 7 illustrates the enactement of the adaptiveness process. When a service/co-engine detects a failure, it injects the updates rules in the subsolution of the common output T_{out} and in the sources T_{in} at runtime. As discussed in Section 2.4, the workflow will resume automatically.

4 Related work

In contrast to classical workflow manager systems, approaches using the chemical metaphor [2, 4, 9] envisage the workflow execution as an autonomous process evolving in time according to the requirements and dependencies without bounding to any preset constraint. For instance, the work presented in [8] represents the enactment as an abstract coordination modelling which identify different levels of the workflow definition process: an abstract workflow that expresses the logic, and a concrete workflow where logical entities are assigned to resources for execution. They propose a uniform declarative formalism to describe workflows and coordination strategies. However, they do not exploit higher-order constructs nor envisage full dynamicity at runtime to support complex mechanisms such as fault-tolerance.

Another chemical model [2] defines a general workflow notation, including common workflow constructs, separating data and control flow. This formalisation pays special attention to the resource instrumentation through its identifiers to ensure dynamic coordination. However, they do not address a decentralised execution. In the same direction, a service-based application

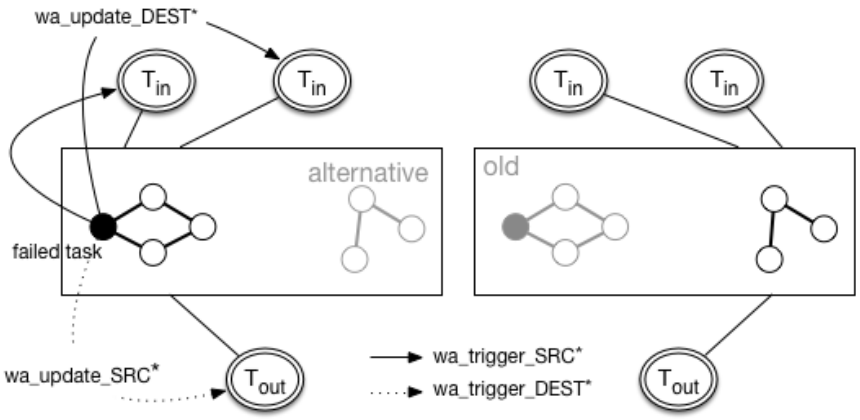


Figure 7: Adaptiveness implementation

is introduced in [4], modelling the process of associating service functionalities with required conditions as an evolving and always running middleware mechanism. They underline the support of environmental changes by taking into account QoS requirements. Both approaches show the interest of tackling, at the workflow definition level, the adaptation of the service instantiation, but they do not associate that adaptation to the semantics of the workflow definition.

Our architectural approach takes its roots in the work presented in [6]. Nevertheless, that previous work does not address data iteration strategies, and neither does adaptiveness. The GinFlow prototype also leverages workers for task parallelism, thus boosting its scalability in contrast to the implementation presented in [6].

Most of workflows manager systems initiatives achieve the required enactment flexibility by means of the infrastructure (e.g., re-submissions strategies [7], pilot jobs [3]). On the other hand, Tolosana-Calasanz et al. [12] propose an adaptive exception handling for scientific workflows at definition level. Their work proposes two patterns to manage the exception handling based on the Reference Nets-within-Nets formalism: propagation and replacement. In spite of mechanisms for dynamically adapting the workflow structure at runtime without having to be aware of the underlying infrastructure, the resulting representation with their reference model suggests a quite complex workflow definition, where the original scenario and the alternative path are mixed.

5 Conclusions

In this paper, we have presented a set of programming abstractions based on a chemical metaphor to express data-driven workflows. We introduced an extension of the HOCL language aiming at easing the development of such programs. We have shown how to, out of the mere description of the workflow, it can get enabled in a decentralised environment. Also, we devised a set of abstractions (and their implementation) to automate exception-handling, guided by the user at design time. The GinFlow prototype is currently under development and testing. In parallel to this work, the GinFlow prototype is currently integrated to the TIGRES workflow tools [11].

References

[1] J.-P. Banâtre, P. Fradet, and Y. Radenac. Generalised multisets for chemical programming. *Mathematical Structures in Computer Science*, 16(4):557–580, 2006.

[2] M. Caeiro, Z. Németh, and T. Priol. A chemical model for dynamic workflow coordination. In *19th Euromicro International Conference on Parallel, Distributed and network-based Processing*, PDP 2011, Ayia Napa, Cyprus, February 2011.

[3] A. Casajus, R. Graciani, S. Paterson, and A. Tsaregorodtsev. DIRAC pilot framework and the DIRAC workload management system. *Journal of Physics: Conference Series*, 219(6), 2010.

[4] C. Di Napoli, M. Giordano, Z. Németh, and N. Tonellotto. Adaptive instantiation of service workflows using a chemical approach. In *16th International Euro-Par Conference on Parallel Processing*, Euro-Par 2010, Ischia, Italy, August 2010.

[5] P. Dittrich, J. Ziegler, and W. Banzhaf. Artificial chemistries—A review. *Artificial Life*, 7(3):225–275, 2001.

[6] H. Fernández, C. Tedeschi, and T. Priol. Rule-driven service coordination middleware for scientific applications. *Future Generation Computer Systems*, 35:1–13, 2014.

[7] D. Lingrand, J. Montagnat, and T. Glatard. Modeling user submission strategies on production grids. In *18th ACM International Symposium on High Performance Distributed Computing*, HPDC'09, Munich, Germany, June 2009.

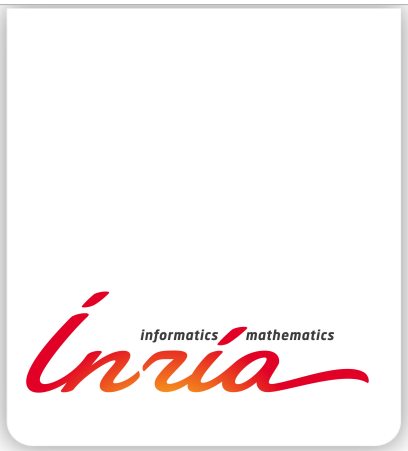
[8] Z. Németh, C. Pérez, and T. Priol. Workflow enactment based on a chemical methaphor. In *3rd IEEE International Conference on Software Engineering and Formal Methods*, SEFM'05, Koblenz, Germany, September 2005.

[9] Z. Németh, C. Pérez, and T. Priol. Distributed workflow coordination: molecules and reactions. In *IEEE IPDPS*, Rhodes Island, Greece, April 2006.

[10] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.

[11] J. Rojas Balderrama, M. Simonin, C. Morin, H. Valerie, L. Ramakrishnan, A. Deborah, and C. Tedeschi. Combining Workflow Templates with a Shared Space-based Execution Model. In *9th Workshop on Workflows in Support of Large-Scale Science*, pages 50–58, New Orleans, USA, November 2014.

[12] R. Tolosana-Calasanz, J. A. Bañares, O. F. Rana, P. Álvarez, J. Ezpeleta, and A. Hoheisel. Adaptive exception handling for scientific workflows. *Concurrency and Computation: Practice and Experience*, 22(5):617–642, 2010.



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399